UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

BERNARDO BELTRAME FACCHI

RANDOM ELIXIR CODE GENERATION APPLIED TO COMPILER TESTING

CHAPECÓ
2024

BERNARDO BELTRAME FACCHI

RANDOM ELIXIR CODE GENERATION APPLIED TO COMPILER TESTING

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal da Fronteira Sul (UFFS), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Prof. Dr. Samuel da Silva Feitosa

Este Trabalho de Conclusão de Curso foi avaliado e aprovado pela banca avaliadora em: 28/06/2024

BANCA AVALIADORA

Prof. Dr. Samuel da Silva Feitosa - UFFS

Prof. Dr. Andrei de Almeida Sampaio Braga - UFFS

André Rauber Du Bois - UFPel

Random Elixir Code Generation Applied to Compiler Testing

Bernardo Beltrame Facchi
bernardobf[at]outlook.com.br
Universidade Federal da Fronteira Sul
Chapecó, SC, Brazil

Andrei de Almeida Sampaio Braga
andrei.braga[at]uffs.edu.br
Universidade Federal da Fronteira Sul
Chapecó, SC, Brazil

André Rauber Du Bois
dubois[at]inf.ufpel.edu.br
Universidade Federal de Pelotas
Pelotas, RS, Brazil

Samuel da Silva Feitosa
samuel.feitosa[at]uffs.edu.br
Universidade Federal da Fronteira Sul
Chapecó, SC, Brazil

ABSTRACT

Developers expect compilers to be correct. Unfortunately, these tools are not entirely bug-free. A failure introduced by the compiler could compromise a critical system and consequently have catastrophic consequences, specially in applications of great complexity, affecting both end users and developers. Such failures can lead to significant financial losses, security vulnerabilities, and a loss of trust in the software's reliability. Therefore, testing and validating all the compiler functionalities to assure its correctness is essential given their importance in software development. In light of the given context, this paper describes a random code generation tool using Haskell that generates well-typed Elixir code by adhering to a specified syntax and typing rules, which serves as input for property-based tests, striving to contribute to the overall quality and dependability of software systems built using Elixir.

CCS CONCEPTS

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

KEYWORDS

Code generation, Elixir Compiler, Property-based Testing

ACM Reference Format:
Bernardo Beltrame Facchi, Andrei de Almeida Sampaio Braga, André Rauber Du Bois, and Samuel da Silva Feitosa. 2018. Random Elixir Code Generation Applied to Compiler Testing. In *Proceedings of Brazilian Symposium on Programming Languages (SBLP '24).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The Elixir programming language has rapidly emerged as a powerful tool in the landscape of modern software development, known for its scalability, concurrency, and functional programming capabilities based on the lambda calculus [2]. Its syntax, influenced by Ruby, offers a user-friendly experience while maintaining the efficiency and reliability needed for complex, real-time applications. This combination of usability and performance has led to its increasing adoption in various domains, from web development to embedded systems.

With the rise of popularity of the Elixir programming language and its adoption in many major projects, it is important to have mechanisms to ensure that the compiler works correctly. Assuring that the machine language code runs accurately to what was written by the programmer is essential to software development and for the language's usage since programmers do not want failures to be introduced in their software during the compilation process.

A common approach to test compilers is to write and execute case tests manually. However, due to the complexity of modern compilers, and the time-consuming task of crafting test cases, such an approach is often not efficient enough to assure the correctness of the compiler comprehensively, since this method can easily overlook edge cases and rare scenarios that could cause the compiler to behave unexpectedly. Random code generation addresses this limitation [11] by automating the creation of diverse and extensive test cases. The development of a tool to exhaustively generate test cases allows us to cover a larger subset of the language and test various functionalities of a compiler in a much more efficient and systematic way.

However, developing a random code generation tool presents a significant challenge since several constraints, imposed by the language's compiler, must be adhered to in order to generate valid and useful test cases, such as the syntactical correctness and type system requirements. These restrictions guarantee that the generated code will be valid, so it can then be used as input to the compiler and allow for the application of property-based testing, exhaustively testing the generated code in an automated manner.

We investigate random code generation using a bottom-up, goal-oriented approach [1, 6, 11] to generate randomized programs. We implement a random code generator using Haskell, a language well-suited for code verification due to its strong static type system and emphasis on pure functions, capable of generating type-correct expressions based on a subset from the Elixir language. To measure the quality of our generator, we check the code coverage to verify if our generator covers a wide range of code scenarios and use property-based testing with the QuickCheck library [7] to check if the generated code is well-typed.

SBLP '24, September 30 – October 04, 2024, Curitiba, PR

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

The remainder of this text is as follows: in Section 2, we describe, shortly the Elixir programming language along with its syntax and typing rules. The steps to generate Elixir programs are described in Section 3. Section 4 presents our prototype implementation and describes the analysis of the generated code. Related works are in Section 5, and lastly, Section 6 provides a conclusion and suggestions for future work.

2 ELIXIR LANGUAGE

The Elixir functional programming language, created by José Valim, had its first version released to the public in 2014, and runs on top of the Erlang Virtual Machine (BEAM). Elixir offers productive programming for secure and maintainable distributed applications by leveraging the virtual machine resources on which it is based [12]. Elixir is functional, process-oriented, scalable, concurrent, and fault-tolerant [5].

Functional programming, which Elixir is based upon, promotes a programming style that helps programmers to write short, concise, and maintainable code.

Elixir is a process-oriented language where code runs inside lightweight threads (called processes) that are independent of one another and exchange information amongst themselves via messages. Due to their lightweight nature, it is possible to run hundreds of thousands of processes concurrently in the same machine (vertical scaling) and also on different connected machines (horizontal scaling).

Elixir can react to possible failures through supervisors, who describe how to restart specific system parts, returning to a known initial state guaranteed to work. Therefore, Elixir is an excellent choice for event-driven systems and robust architectures.

The subsequent sections provide an explanation of the Elixir syntax considered for this paper and its corresponding type system.

2.1 Syntax

Most Elixir constructors are syntactic sugar based on function application and pattern matching. Types in Elixir are polymorphic, set-theoretic, and recursively defined [2]. The syntax this project was based upon is described in Figure 1.

The base types are defined by integers, atoms, functions, and tuples. The top type ($\mathbb{1}$), the type of all values, is defined as $\mathbb{1} = \text{int} \vee \text{atom} \vee \mathbb{1}_{fun} \vee \mathbb{1}_{tup}$, and the bottom type $\mathbb{0}$ as $\mathbb{0} = \neg\mathbb{1}$, which correspond to Elixir's term() and none() types, respectively.

Types contain base types, constants (representing singletons), type variables, the constructors $\overline{t} \rightarrow t$ ($\overline{t}$ denotes the sequence $t_1 \dots t_n$) and $\{\ \overline{t}\ \}$ for functions and tuples, respectively, and two connectives, union ($\vee$) and negation ($\neg$).

Expressions have constants, variables, lambda functions, function applications, tuples (and their projections), let, and case. The let expression expresses that the result of one expression applied to another will be assigned to one variable. The case expression expresses code branching from an expression through pattern matching.

Patterns can be variables, constants, or a tuple of patterns. Guards are boolean tests composed of guards and selectors. Selectors are the building blocks of guards, which can be a variable, a singleton, an element, or a tuple size.

Syntax

$b ::=$	Base types
$\quad \text{int} \mid \text{atom} \mid \mathbb{1}_{fun} \mid \mathbb{1}_{tup}$	
$t, s ::=$	Types
$\quad b \mid c \mid \alpha \mid \overline{t} \rightarrow t \mid \{\overline{t}\} \mid t \vee t \mid \neg t$	
$e, f ::=$	Expressions
$\quad c \mid x \mid \lambda(\overline{x}.e) \mid f(\overline{e}) \mid \{\overline{e}\} \mid \text{elem}(e, e) \mid e + e$	
$\quad \text{let } x : t = e \text{ in } e \mid \text{case } e \text{ do } \overline{pg \rightarrow e}$	
$p ::=$	Patterns
$\quad x \mid \_ \mid c \mid \{\overline{p}\}$	
$g ::=$	Guards
$\quad g \text{ and } g \mid g \text{ or } g \mid \text{not } g \mid \text{is\_integer}(d)$	
$\quad \text{is\_atom}(d) \mid \text{is\_tuple}(d) \mid \text{is\_function}(d, d)$	
$\quad d == d \mid d \neq d \mid d < d \mid d \leq d$	
$d ::=$	Selectors
$\quad c \mid x \mid \text{elem}(d, d) \mid \text{tuple\_size}(d)$	

Figure 1: Syntax of Elixir [2] considered for the generation process.

2.2 Type System

Both statically and dynamically typed programming languages have a type system. Its purpose is to define how the language constructions can be used besides its grammar. This process is carried out through a set of rules. Figure 2 presents a formal subset of the Elixir type system [2].

$$\frac{}{c : c}\ (\text{cst}) \qquad \frac{\Gamma \dashv x : t}{x : t}\ (\text{var})$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}\ (+) \qquad \frac{\overline{x} : \overline{s} \vdash e : t}{\lambda(\overline{x}.e) : \overline{s} \rightarrow t}\ (\lambda)$$

$$\frac{f : \overline{s} \rightarrow t \quad \overline{e} : \overline{s}}{f(\overline{e}) : t}\ (\text{app}) \qquad \frac{\overline{e} : \overline{t}}{\{\overline{e}\} : \{\overline{t}\}}\ (\text{tuple})$$

$$\frac{f : \text{int} \quad e : \{t_0, \dots, t_n\}}{\pi_f(e) : t_i}\ (\text{proj}) \qquad \frac{f : s \quad x : s \vdash e : t}{\text{let } x : s = f \text{ in } e : t}\ (\text{let})$$

$$\frac{e_1 : s \quad \Gamma, \overline{\text{vars}(p, s) \vdash e_2} : t}{\Gamma \vdash \text{case } e_1 \text{ do } \overline{pg \rightarrow e_2} : t}\ (\text{case})$$

Figure 2: Type system considered for expressions.

Each rule determines how the type of a specific term should be verified by the compiler. A term is valid if its premises are in accordance with the type system restrictions. Considering the approach of generating code by following each rule, we guarantee that the randomly generated code will be correct, allowing the code to be compiled and executed.

The type system shown in Figure 2 presents the typing rules we consider during the development of the random generator. The definition of each rule is described as follows:

Constants. The rule (cst) simply defines a constant and has no premises.

Variables. Rule (var) specifies that a variable x will be evaluated to a type t if a variable x of type t exists in the Γ context.

Additions. Rule (+) specifies that the sum of two expressions will be of type *int* if these two expressions are of type *int*.

Lambdas. Rule (λ) specifies that a lambda expression with parameters $\overline{x}$ and an expression e has a type $\overline{s} \rightarrow t$ if expression e has a type t considering a context extended with $\overline{x}$ of type $\overline{s}$.

Applications. The rule (app) defines that the invocation of a function f with parameters $\overline{e}$ will be of type t if f is a function of type $\overline{s} \rightarrow t$, and parameters $\overline{e}$ are of type $\overline{s}$.

Tuples. Rule (tuple) specifies that a tuple $\overline{e}$ will be evaluated to tuple $\overline{t}$ if the sequence of expressions $\overline{e}$ has the sequence of types $\overline{t}$.

Projections. The rule (proj) defines that given an index f and an expression $e \rightarrow \{\overline{t}\}$, the evaluated type will be the type found in the tuple of the given index. The projection is not a type-safe expression, since the projection index can be the result of an expression, the type system cannot statically guarantee that the projection will be limited to the tuple's size, which may raise an index out range.

Bindings. The rule (let) defines that a *let* expression is of type t when the expression f is of type s and expression e is of type t considering an extended context with variable x of type t.

Alternatives. The rule (case) defines that a *case* expression is of type t given that expression e_1 is of a valid type s, and all expressions of e_2 have a type t, considering that a context of variables is extended with variables and types extracted from the pattern p[1].

Considering these guidelines, we propose a method to generate well-typed Elixir programs, the specifics of which are elaborated in the following section.

3 CODE GENERATION

The process of generating random, well-typed Elixir code is divided into three key steps: (*i*) randomly generating a valid Elixir type, (*ii*) randomly generating an expression in an abstract representation, and (*iii*) compiling the generated expression into Elixir concrete syntax. On this basis, the generated type is used as input to the expression generation process, which uses the typing rules as constraints to create valid expressions based on a bottom-up approach, where to satisfy the conclusion of a rule, it is necessary to respect their premises. This process gives rise to an expression generation judgment, as follows.

Definition 1. *Expression generation judgment.* $\Gamma; T \overset{exp}{\Longrightarrow} e$

Given a Γ context containing the free variables, and a type T, a new expression e is generated by selecting a syntactical constructor at random respecting the typing rules.

The generation technique we use is derived by interpreting the typing rules from Figure 2 in reverse order. In essence, to produce an expression that appears as the result of a rule, one must initially create expressions that form the rule's premises and then merge them. Consequently, the process of generating a term may recursively need the creation of sub-goals. Employing the typing rules guarantees that the generated terms are correctly typed.

To convert the generated expressions to Elixir code, we had to adhere to the Elixir's syntax, translating the generated AST into the concrete syntax. Therefore, we implemented a *show* instance in Haskell for each data type. The *show* instance allows us to dictate how each data type should render as a string. Thus, by using string manipulation to adhere to Elixir's syntax, we ensure that the generated code is syntactically correct and compatible with the Elixir compiler, allowing the output from our generator to be compiled and executed in an Elixir environment.

3.1 Souce code

The source code presented in this article was developed in Haskell (version 8.6.5), using the QuickCheck library (version 2.12.6.1) for property-based testing. Throughout the text, only excerpts of the code that are important for understanding the generation mechanism are presented, omitting parts that may distract the reader from the high-level implementation understanding. All the code produced for this article can be found in the GitHub repository of the work[2].

The subsections that follow present in detail how types and expressions are randomly generated considering the language constraints, i.e., the syntax and typing rules.

3.2 Type Generation

The first step was to define all the valid Elixir types according to the language syntax in Haskell using Algebraic Data Type (ADT) constructors. We implemented the primitive types (*int* and *atom*), tuple type and function type, as described in the next piece of code.

```
data PrimType = IntType
              | AtomType

data Type     = PType PrimType
              | FunType [Type] Type
              | TupleType [Type]
```

The process of generating types is entirely based on syntax, which implies that it's unrestricted and any valid type can be generated at random. We achieve this by using the `frequency` function from the QuickCheck library, which allows the use of weights to define the frequency in which each function will be selected at random. The type generation follows a recursive process. If the selected type is final, i.e., a primitive type, the return is immediate. Otherwise, we recursively continue the generation. To avoid non-termination due to the recursive approach, we decrement a fuel (the *s* parameter) on each recursive call so that when it reaches zero, only terminal types can be created, forcing the generation to stop. The function presented next shows how types are generated at random.

```
genType :: Int -> Gen Type
genType s | s > 0 = frequency [
```

[1]For simplicity, we refer to the function *vars* which is responsible for extracting each variable with its respective type from a given pattern.

[2]https://anonymous.4open.science/r/Elixir-Generator-840F

```
                (2, genPrimType),
                (1, genTupleType s'),
                (1, genFunType s')
                ]
    | otherwise = genPrimType
    where s' = s `div` 2
```

Note that, if the parameter s is greater than zero, all the types can be generated, otherwise the generation function restricts the process to generate only primitive types.

3.3 Expression Generation

The expression generation process is similar to the type generation process. The difference is that it must be guided by the typing rules presented in Figure 2 to generate type-correct expressions. The expressions we defined using Haskell ADT constructors are presented next.

```
data Expr = Var String
          | Tuple [Expr]
          | TupleElem Expr Expr
          | Case Expr [Alt]
          | Plus Expr Expr
          | App Expr [Expr]
          | Lambda [(String, Type)] Expr
          | Literal Literal
          | Let String Expr Expr
data Alt = CaseAlt (Maybe Pattern) (Maybe Guard) Expr
```

The objective of the expression generation is to create a well-typed expression at random that should be evaluated to a specific type. To guarantee that the expression is well-typed, we respect the restrictions imposed by the typing rules during the generation process. For this reason, to generate an expression of a specific type, only a subset of the typing rules can be considered. For example, the tuple rule can only be used when the expected type is a tuple, the lambda rule can only be used when the expected type is a function type, the arithmetic addition rule can only by used when an integer type is expected, and so on.

As shown by the process generation judgment, for the generation of each expression, it is expected the use of two inputs: (1) a context (initially empty) containing variables that might be used during the generation of sub-expressions[3] that is fed during the generation process with new variables when they are created; and (2) a type, that defines what the expression should be evaluated to. Note that during the generation of sub-expressions their return type might not be the same as the return type of the initial expression.

The generation process uses a bottom-up approach, where to generate an expression, we must satisfy the expression typing rule's premises, which might require the generation of sub-expressions. Hence, the generation method explored in this paper is recursively defined and guided by the type system. This approach guarantees that the expressions created are well-typed.

To understand how the typing rules are used to guarantee the generation of type-correct expressions, let's consider the following example.

[3]A sub-expression is part of an expression that is by itself an expression.

Example 1. *Using the expression generation judgment to create a new expression of type int.* $\Gamma; int \overset{exp}{\Longrightarrow} e$

A typing rule can be formatted using the question mark ? as a placeholder for that expression, representing the first generation step, as follows:

$$\Gamma \vdash ? : int \tag{1}$$

Suppose the rule for arithmetic addition (+) was selected at random. Then, the second generation step would look like:

$$\frac{\Gamma \vdash ?_1 : int \qquad \Gamma \vdash ?_2 : int}{\Gamma \vdash ?_1 + ?_2 : int}\ (+) \tag{2}$$

The question marks ($?_1$ and $?_2$) represent the sub-expressions that will be generated as sub-goals.

To generate each sub-goal, the generation judgment should be used recursively for each placeholder. It means that other typing rules can be selected.

Suppose that, for short, each sub-goal selected the rule for constants (cst). Then, the third generation step would be:

$$\frac{\dfrac{}{2 : int}\ (\text{cst}) \qquad \dfrac{}{5 : int}\ (\text{cst})}{\Gamma \vdash\ 2 + 5 : int}\ (+) \tag{3}$$

As can be noted, since only terminal rules (without premises) were selected, the generation process finished producing a new expression $(2 + 5)$ of type int.

Next, we explain in detail how each expression is generated according to the presented expression generation judgment.

Literal generation. A literal expression is simply the value of either an integer or an atom, following the rule `(cst)`. Note that this expression is final. The value of this expression will be either a random integer generated by the QuickCheck library using the `arbitrary` function or a random atom selected amongst a list of previously declared atoms.

Variable generation. To generate a variable expression, according to the `(var)` rule, the generator has to look through the given context and randomly select an element x that matches with the required type t. It is worth mentioning that this expression can only be generated if the context of variables is not empty and we have at least one variable whose type is the same as the required type. This expression is final and thus it doesn't need to generate sub-expressions.

There is an important observation about our algorithm that should be mentioned here. The process of generating random expressions is not completely random, i.e., it doesn't select any syntactical constructor that could generate an expression of a given type at random, because it might take an expression that could not be completed to be type-correct. This would require the recursive process to backtrack, looking for another expression that could be fulfilled. Instead, the generation judgment produces a list of valid candidate expressions to be selected, considering only a subset of the typing rules. With this approach, we are able to guarantee the generation of well-typed expressions will be finished, and also avoid the need for backtracking.

Arithmetic addition expression. According to the type system, through the rule (+), the expression that produces a sum of two other sub-expressions $e_1 + e_2$, will only be well-typed if both sub-expressions are evaluated to the *int* type[4]. Therefore, when the generator creates such an expression, it must generate two sub-expressions of type *int* recursively. Next, we can see an example of generated code.

Example 2. *An expression that uses the arithmetic addition produced by our generator.*

```
9 + fn x0, x1 -> x1 end.(:var196, 6)
```

Note that this example has an expression with the sum operator, where the function application that appears on the right-hand side can be seen as a sub-expression. Any expression can be generated to fulfill the rule's premises, as long as its evaluation is of the *int* type. Although such kind of code might look unusual for a regular developer, it is useful to reach all the branches on the compiler.

Lambda expression generation. Following the typing rule (λ), to generate a well-typed lambda expression $\lambda(\overline{x}.e)$ of type $\overline{s} \rightarrow t$, first we need to generate a sequence of parameter names $\overline{x}$, and then generate the function body e using the expression generation judgment recursively, considering a context augmented with the generated variables $\overline{x}$ with type $\overline{s}$. Next we can see an example of generated code.

Example 3. *A lambda expression that uses the arithmetic addition produced by our generator.*

```
fn x0, x1 -> x1 + x0 + 374 end
```

This example shows that a function with type $int, int \rightarrow int$ was generated, containing two addition operations in its body expression. Although being a simple example, it exhibit a characteristic of our generator. When creating a body expression, the generation judgment can be invoked recursively several times, one for each sub-expression. Since the algorithm is recursive, it can generate a variety of functions containing any sorts of the syntactical construtors.

Function application generation. To generate a well-typed function application, the algorithm follows the rule (app). The input for the expression generation judgment is a type t, which should guide the type of the generated expression. An application is defined by having two sub-expressions. The first sub-expression f should evaluate to a function, and the second is sequence of sub-expressions $\overline{e}$, representing the actual parameters to be applied to that function. The generation process starts by generating a sequence of types $\overline{s}$ to define the parameter types. Then, an expression of type $\overline{s} \rightarrow t$ is generated recursively. After that, the expression generation judgment is applied recursively to generate each expression in $\overline{e}$, considering the types defined in $\overline{s}$. That way, we guarantee that all parameters expressions are defined with correct types. The next example shows another result of the generator.

Example 4. *A function application generated by our algorithm.*

```
fn x1, x2 -> x1 end.(
  case :var202 do
    :var1 when is_tuple(:var17) -> 2 + 3
    x1 when 9 <= 9 -> 8
    _ -> 2
  end,
  :var53
)
```

In this example, the generator created a lambda expression with two formal parameters ($x1$ and $x2$) and, consequently, two expressions to supply the function with the actual parameters. The value used to supply the first parameter will be the result of evaluating the *case* expression, which was recursively generated, and the value to supply the second argument is a random generated *atom*.

Tuple generation. To generate a tuple expression, the expression generation judgment follows the rule (tuple), where a sequence of expressions $\overline{e}$ should be generated according to a sequence of types $\overline{t}$. The following example shows an example of a tuple generated by our algorithm.

Example 5. *A tuple expression with different types produced by our generator.*

```
{-15, :var54, fn x -> -18 + x end}
```

The example above shows a tuple generated from the type $int, atom, int \rightarrow int$, where an *int* -15, an atom :var54, and a function fn x -> -18 + x end were generated, respectively to the required tuple type. One can note that only final expressions appear on this example. This happened because the fuel variable that limits recursion was set to zero to simplify the comprehension of the example. However, any expression could be generated inside a tuple, final or not, as long as its evaluation was of the specified type for each index.

Projection generation. The tuple projection expression is generated according to the rule (proj), and receives as input a type t_i of which the projected element should belong to. The projection (function elem) is composed of two expressions (f and e). The first expression f is a literal of the *int* type and represents the index of the desired tuple element. The second expression e should be an expression that evaluates to a tuple type. Expression f is generated as an integer between one and a maximum threshold, which defines the maximum size of the generated tuple. Then, expression e is generated with size at least the value of f. Besides this step, we have to guarantee that the projection of index f on e has type t_i.

Example 6. *A projection expression produced by our generator using as input the type int.*

```
elem({:var105, -14, -28, 30, -29}, 2)
```

The example above shows a tuple projection expression with a tuple of random elements containing only final expressions, and an index to be projected. It is worth mentioning that we ensure that the tuple element at the given index will be of the correct type by forcing the type of the element at the given index to be the same as the expression generation judgment input type before generating the tuple expression.

[4]This is a restriction imposed by the considered formalization of the Elixir type system [2]. The complete language allows using the sum operator polymorphically with numeric types.

Let binding generation. To generate a *let* expression, the algorithm follows the rule (let). The generation judgment receives as input a context of free variables, and a type t. A *let* binding is created by generating a random name x with a type s, an expression f of type s, and an expression e which should have type t, considering an extended Γ environment with the variable x of type s.

Example 7. *A let binding generated by our algorithm.*

```
x0 = fn x0 -> x0 + x0 end.(
  case :var208 do
    :var233 when 18 < -16 -> 29
    x0 -> 0
    _ -> -11
  end
)
```

Note that the concrete syntax used to represent the let expression shown in the example above is different from the one shown in Figure 1 since the syntax on which we based our generator is not fully implemented in Elixir yet [2]. Nevertheless, the (let) rule constraints shown in Figure 2 are still respected, and the generated let expressions can be compiled and executed by Elixir as usual with the same semantics.

Case generation. Elixir provides pattern matching, which allows a developer to assert on the shape or extract values from data structures, and to augment it with guards to perform more complex checks. In the subset we are working on, we can apply pattern matching using a *case* expression.

As we can see on the Elixir syntax and typing rules, the *case* expression involves, besides expressions and types, the use of patterns p, and guards g. To represent the abstract syntax of both, we used Haskell ADTs, as we can see below.

```
data Pattern = VarPattern String
             | WildcardPattern
             | LiteralPattern Literal
             | TuplePattern [Pattern]

data Guard = GuardAnd Guard Guard
           | GuardOr Guard Guard
           | GuardNot Guard
           | GuardIsInt Sel
           | GuardIsAtom Sel
           | GuardIsTuple Sel
           | GuardEqual Sel Sel
           | GuardNotEqual Sel Sel
           | GuardLess Sel Sel
           | GuardLessEqual Sel Sel
```

This subset allows to pattern match variables, wildcards, literals and tuples. We can augment the pattern with conditional guards which allow us to use several boolean and relational operators, besides some extra functions. We defined specific generators for patterns and guards from a given type, similarly to what was done for types[5].

[5]We omit the details of these generators for space reasons.

Having generators for patterns and guards, we can proceed for the *case* expression generation. This expression is somewhat intricate to define. The algorithm follows the rule (case), and receives a type t as input. On the first step, it is necessary to generate an expression that is allowed to be pattern matched[6]. So, we generate a type s that can be used with pattern matching at random, and then an expression e_1 is created using the expression generation judgment with that type s as input. After that, the algorithm generates a sequence of case alternatives $\overline{pg \rightarrow e_2}$. To avoid generating overlapping and non-exhaustive patterns, we opted for a fixed size of three alternatives for each case expression. The first would consider patterns p that could be refuted, including the literal and the tuple pattern. The second pattern matches with a variable. And the last is used with a wildcard, acting as a default fail-safe alternative, to ensure that the case expression will always match something and return. It is important to mention that, conventionally, a variable pattern would always be evaluated as true. However, due to the guards g, that might not be always the case. The use of guards is optional, so we use a weight to define the frequency the guard should be generated. Besides all that, for each alternative the algorithm has yet to generate an expression which respects the input type t. Respecting all these constraints, we were able to generate well-typed *case* expressions.

Example 8. *A case expression with guards produced by our generator.*

```
case -9 do
  11 -> -19;
  x0 when is_tuple(elem({:var13, -25}, 1)) -> x0 + -8;
  _ -> 38
end
```

On the example, we can note that the second alternative is not evaluated as *true* due to the generated guard, because the result of the (elem) expression is not a tuple, requiring the generation of a third alternative that should always be evaluated as *true*.

The method described in this section has demonstrated a technique for creating correctly typed expressions in accordance with the typing rules specified by Castagna, Duboc, and Valim [2]. The last is the main designer of the Elixir language. The next section shows some experiments conducted using a prototype version of our algorithm, aiming to verify its effectiveness in producing valid code that is accepted by the compiler. Given that the code produced conforms to a valid Elixir program, we are confident that the approach and its implementation are suitable for testing scenarios.

4 PROPERTY-BASED TESTS

Property-based testing is a software testing methodology that systematically explores the behavior of software programs by validating if a property holds for a wide range of automatically generated inputs. Unlike traditional example-based testing, where specific input-output pairs are manually defined, property-based testing defines high-level properties the code should satisfy universally.

A key strength of property-based testing lies in its ability to expose edge cases and hidden bugs that might remain undetected with traditional testing techniques. By generating a vast array of random

[6]In Elixir, we cannot apply pattern matching in expressions of a function type.

inputs, property-based testing ensures the software is tested far beyond the limited cases a developer might think to test manually. The goal is to rigorously verify the correctness of the software by exhaustively testing it across diverse and unanticipated input scenarios.

To carry out property-based tests, we used the QuickCheck library, which was developed in Haskell by Claessen and Hughes, with the premise that, instead of tests being generated manually for a specific target code, an arduous and tedious task, generated tests can be tested quickly [7] using such a test generator.

To provide a proof-of-concept of our work, we implemented an Elixir random code generator tool using Haskell based upon the definitions presented in Section 3 and a simple test suite also using Haskell and QuickCheck. To ensure the validity of our generated Elixir code, we implemented a property designed to verify that all generated test cases compile successfully. This property establishes that the code generated by our tool is both syntactically valid and correctly typed. For each generated program, the process involves writing the code to disk and subsequently invoking the Elixir compiler to compile the program and report back whether the compilation was successful or if any error was found.

The property being tested begins by generating a random Elixir program according to the presented syntax and typing rules, as described. This program is then written to a temporary `.ex` file by the Haskell `writeFile` function, ensuring that each test case is isolated and manageable. The next step involves invoking the Elixir compiler to execute the `elixirc` command on the generated file. This command attempts to compile the program and reports whether the compilation was successful along with the program output. We capture the exit status of the compilation process. In the event of a failure, the compiler's error messages are analyzed, providing insights into the specific issues encountered, and allowing us to iteratively refine the generator to ensure that a valid program is generated.

This property-based approach left us confident that each generated test case is valid, and in addition it also provided us with a mechanism for continuously validating the overall effectiveness and reliability of our code generator. The tested property ensures that our tool consistently produces valid, compilable code, which is a fundamental prerequisite for any further semantic testing or execution of the generated programs.

We used the test suite to run ten batches of 1000 tests. Each batch generated, compiled and executed all programs in roughly 6 minutes on a computer with an Intel(R) Core i7-10700k CPU (5,00 GHz x 8) running Ubuntu 20.04.6 LTS on Windows 11 through WSL. It is worth noting that all generated programs compiled successfully, confirming that we are indeed generating only well-typed programs, meaning that our code generation and testing framework were reliable in validating the generated Elixir code. It indicates that the generated programs can be used in other test scenarios.

We also generated programs and compared them across different versions of Elixir by looking at their outputs. We conducted ten sets of 1000 tests, each taking around 18 minutes. The versions we used were 1.15.0, 1.16.3, and 1.17.1, all of which utilized Erlang OTP/25. We employed asdf to manage multiple Elixir versions. During test execution, no differences in output were found between the versions we tested. Some tests, however, could not be completed due to the compilation process taking more than 10 seconds, which forced us to reduce the generated code size.

Additionally, we employed the Haskell Program Coverage (HPC) tool to assess the diversity of the generated programs and provide detailed insights into the execution paths taken by our code generation logic. Given that our approach to code generation is randomized, we have no control over what branches the algorithm will take during execution, it is crucial to ensure that this randomness adequately explores all syntactical constructs and does not inadvertently miss any critical paths or edge cases.

Upon analyzing the statistics report provided by HPC, we observed that 100 percent of the syntactical constructors were covered in a batch of 1,000 test cases. This result is significant as it demonstrates that our random code generation method effectively explores all possible code patterns within defined constraints. Achieving full coverage means that our tool can generate a wide variety of valid Elixir programs, ensuring that no syntactical edge cases are overlooked, which implies that our program generation approach can be trusted to test the full spectrum of valid Elixir syntax.

By ensuring that all syntactical constructs are represented in the generated programs, we can state that our tool provides good inputs for testing purposes, which is important for further tests, increasing the probability of identifying and addressing potential bugs.

5 RELATED WORKS

Although the concept of random code generation originated in the early 1960s [13], it continues to be a challenge to this day, even with many advancements in the field, since it is hard to create a generator that respects the constraints imposed by compilers. The studies presented in this section explore various techniques for random code generation aimed at testing compilers. These studies have provided insights into how to develop a random code generation tool and how to effectively generate a diverse range of test cases that can thoroughly test the several capabilities of a compiler.

Palka; Claessen; Russo; Hughes [11] tested the Haskell compiler (GHC) by generating lambda terms to locate errors in the target compiler. Ensuring that those errors were not found by end users, thus contributing to the stability and reliability of GHC. The author used the QuickCheck library due to its usefulness in program-based testing properties and, additionally, used a technique called shrinking to reduce the size of the generated code, simplifying it and facilitating its correction process by compiler developers. The authors contribution was notable because it popularized the use of random testing, a technique not that widespread at the time. Despite generating only a limited subset of Haskell, its efforts revealed flaws in the compiler. Similarly to their work, we pretend to explore random code generation with property-based testing to contribute to the stability and reliability of the targeted language.

Livingskii; Babokin; Regehr [9] developed the Yet Another Random Program Generator (YARPGen) for C and C++, which tested the GCC, LLVM, and Intel® C++ Compiler. YARPGen supports code generation policies, which alter the generated code seeking to avoid, or at least delay, the saturation point of the generator, which, to put it simply, is the point where it cannot find more bugs. The test case generated by YARPGen goes through a configurable collection of

compilers and compilation options. If a compiler fails or a difference gets found from all the outputs, the code goes through the shrinking process and goes to a bug classifier developed by the authors. Through YARPGen, the authors found 221 bugs divided between the GCC, LLVM, and Intel® C++ Compiler. Some of these bugs were independently rediscovered by developers of large projects, proving that their tool can find bugs made by programmers. Contrary to the authors, this work does not use code generation policies and code reduction.

Feitosa; Ribeiro; Bois [4] provided a Java program test generator specification using the Featherweight Java (FJ) formalism to generate well-typed programs. The authors developed an interpreter for FJ in Haskell, provided a type-directed heuristic approach to generating random programs, and made use of the QuickCheck library to perform property-based testing as a brief way to check the typing of the generated programs and ensure that all programs compile. The author's technique regarding a code generator developed based on the targeted language type system along with the QuickCheck library usage is similar to this paper.

Yang; Chen; Eide; Regehr [14] developed CSmith, a random code generation tool for the C language targeting the GCC and LLVM compilers. Even though their approach was already employed previously by other authors, they were capable to generate a large subset of the language while ensuring each program had a single interpretation. Over three years, their efforts revealed more than 325 bugs reported to developers. According to them, most of the found bugs caused the compiler to produce incorrect code without warning, highlighting the importance of the random code generation method for testing compilers. Their results have proven that random code generation is a much more promising technique for compiler testing than test suites, the main technique at the time, stating that their code generation tool can find bugs in difficult to reach compiler areas by generating atypical scenarios.

In addition, the domain of AI-driven and machine learning-based generation of test cases has recently become a focal point of interest. Take, for instance, the study by Liu et al. [8], which employs a machine learning method aimed at C compilers, successfully compiling and optimizing 82% of the produced code to detect bugs that cause compiler crashes. In a similar vein, Cummins et al. [3] utilize the same strategy, with a focus on OpenCL compilers. Lyu et al. [10] introduced a coverage-oriented fuzzer specifically for prompt fuzzing, which cyclically creates fuzz drivers to probe untested library code. These studies employ varied code generation techniques, which could be further investigated within the Elixir compiler testing framework.

6 CONCLUSION

In this paper, a random code generator was presented based on a formalization of Elixir's syntax and type system to generate type-correct Elixir programs. We reasoned that the generation method is sound concerning a subset of Elixir's type system, which includes primitive types and operators, several expressions, and pattern matching with guards. Furthermore, we used the QuickCheck library to perform property-based testing, and the HPC tool to produce an analysis of the Elixir code coverage through the generation and execution of the generated programs.

It is worth mentioning that several modifications were made throughout the development phase as bugs were discovered in the generator. For instance, when the output code wouldn't compile, the counterexamples from QuickCheck were instrumental in rapidly detecting and resolving these issues. Additionally, the lightweight approach of property-based testing allowed us to enhance our understanding of the type system's rules and the correct way to apply them in our Elixir random code generator.

In future work, we can expand the algorithm to cover more syntactical constructors and expressions and implement elixir processes with message exchange between them. Also, differential testing can be applied more thoroughly to compare code code output between multiple Elixir versions to assert whether the code generator has the potential to find and understand bugs. Additionally, the same process can be applied with the appearance of other Elixir compilers.

REFERENCES

[1] Elton Maximo Cardoso, Daniel Freitas Pereira, Regina Sarah Monferrari Amorim De Paula, Leonardo Vieira Dos Santos Reis, and Rodrigo Geraldo Ribeiro. 2022. A Type-Directed Algorithm to Generate Random Well-Formed Parsing Expression Grammars. In *Proceedings of the XXVI Brazilian Symposium on Programming Languages* (, Virtual Event, Brazil,) *(SBLP '22)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3561320.3561326

[2] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. The Design Principles of the Elixir Type System. arXiv:2306.06391 [cs.PL]

[3] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[4] Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. 2019. Generating Random Well-Typed Featherweight Java Programs Using QuickCheck. *Electron. Notes Theor. Comput. Sci.* 342, C (apr 2019), 3–20. https://doi.org/10.1016/j.entcs.2019.04.002

[5] Elixir. 2023. Elixir. https://elixir-lang.org/

[6] Samuel Feitosa, Rodrigo Ribeiro, and Andre Du Bois. 2020. A type-directed algorithm to generate random well-typed Java 8 programs. *Science of Computer Programming* 196 (2020), 102494. https://doi.org/10.1016/j.scico.2020.102494

[7] John Hughes. 2016. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*. Vol. 9600. 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[8] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[9] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages. https://doi.org/10.1145/3428264

[10] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. arXiv:2312.17677

[11] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

[12] PetSI. 2018. ELIXIR: uma linguagem de programação brasileira em sistemas distribuídos do mundo. http://www.each.usp.br/petsi/jornal/?p=2459

[13] Richard L. Sauder. 1962. A general test data generator for COBOL. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference* (San Francisco, California) *(AIEE-IRE '62 (Spring))*. Association for Computing Machinery, New York, NY, USA, 317–323. https://doi.org/10.1145/1460833.1460869

[14] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (jun 2011), 283–294. https://doi.org/10.1145/1993316.1993532

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009